

(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号

特開平5-197563

(43)公開日 平成5年(1993)8月6日

(51)Int.Cl.<sup>5</sup>

G 0 6 F 9/45

識別記号

庁内整理番号

F I

技術表示箇所

9292-5B

G 0 6 F 9/ 44

3 2 2 F

審査請求 未請求 請求項の数20(全 17 頁)

(21)出願番号 特願平4-224790

(22)出願日 平成4年(1992)7月31日

(31)優先権主張番号 7 4 1, 2 9 2

(32)優先日 1991年8月6日

(33)優先権主張国 米国 (U S)

(71)出願人 590000400

ヒューレット・パッカード・カンパニー  
アメリカ合衆国カリフォルニア州パロアル  
ト ハノーバー・ストリート 3000

(72)発明者 バントウォール・アール・ラウ

アメリカ合衆国カリフォルニア州ロス・ア  
ルトス・ハイランズ・サークル 900

(72)発明者 マイケル・シュランスキー

アメリカ合衆国カリフォルニア州サニーベ  
イル・シェナンドー 1139

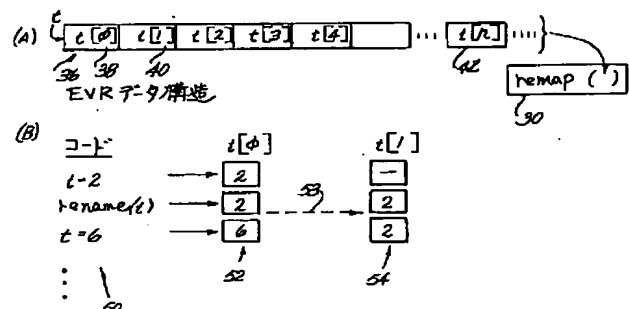
(74)代理人 弁理士 長谷川 次男

(54)【発明の名称】 最適化方法及びコンパイラ

(57)【要約】

【目的】命令レベル・パラレル・プロセッサのロード操作の最適化を行う。

【構成】静的単一代入表現(コードを静的に見た時、各仮想レジスタに1回しか代入が行われないような表現)された中間言語プログラムはループを実行すると1つの仮想レジスタに複数回の代入が行われる。そこで、当該プログラムのどの実行パスにおいても各仮想レジスタに1回しか代入が行われないような動的単一代入表現に変換する。そのため、各仮想レジスタを添字付きレジスタ配列  $t[0], \dots, t[n]$  で表すとともに、この添字と配列の実際の各要素との対応関係を1だけずらす関数  $remap()$  を設ける。ループ中の各仮想レジスタへの代入の前に当該仮想レジスタに  $remap()$  を作用させることにより、同一の添字の仮想レジスタへの代入は1回だけになる。



## 1

## 【特許請求の範囲】

【請求項1】下記のステップ(a)～(d)を有し、命令レベル・パラレル・プロセッサ用コンパイラの間中表現出力を最適化する方法：

- (a) ソース・プログラムを前記コンパイラに供給する；
- (b) 複数の拡張仮想レジスタ・データ構造を生成する：前記データ構造の各々は、個別にアクセスでき、EVR名で識別できる複数のエレメントを有する；
- (c) 前記ソース・プログラムを、拡張仮想レジスタ・データ構造を参照する少なくとも1つの代入分を含む単一代入中間表現に変換する；
- (d) 下記のステップ(d1)～(d3)により、前記単一代入中間表現を動的単一代入中間表現に変換する：
  - (d1) 各代入文に対応して remap() 関数を出力する：remap() 関数は拡張仮想レジスタをパラメータとして受け取る；
  - (d2) 全ての remap() 関数を動的単一代入中間表現中の基本ブロックの先頭に移動する；
  - (d3) 第1の拡張仮想レジスタ・エレメントが第2の仮想拡張レジスタ・エレメントと同じメモリ中のロケーションを参照するか否かをテストし、そうであった場合には動的単一代入中間表現中の第1の拡張仮想レジスタ・エレメントを第2の拡張仮想レジスタ・エレメントで置き換えることによって動的単一代入中間表現中の冗長なロード操作を除去する。

【請求項2】前記拡張仮想レジスタ・データ構造の各々は上限なしに線型に順序付けられたメモリ中の記憶エレメントであることを特徴とする請求項1記載の最適化方法。

【請求項3】前記 remap() 関数は拡張仮想レジスタ・データ構造をパラメータとして受け入れ、前記拡張仮想レジスタの以前には第1の名前を使ってアクセスできた第1のエレメントを第2の名前を使ってアクセスできるようにすることを特徴とする請求項2記載の最適化方法。

【請求項4】前記ステップ(d)は下記のステップ(d4)と(d5)を有することを特徴とする特許請求の範囲だい1項記載の最適化方法：

- (d4) 複数のエイリアス集合を生成する；
- (d5) 前記動的単一代入中間表現中の参照を、同じ配列でかつ同じ添え字についての参照ではない場合を除いて、全て1つの前記エイリアス集合中にストアする。

【請求項5】拡張仮想レジスタ・エレメントの各々は前記中間表現中の添字付き配列変数のテキスト名に結合され、前記結合されたエレメントとテキスト名は複数のマップ・タプルの1つを構成し、前記複数のマップ・タプルは少なくとも1つのマップ・タプル集合中に対応付けられ、

## 2

前記ステップ(d)は、中間表現ステートメントを調べ、当該ステートメントの内容に従ってタプルを集合から消去し、タプルを集合に付加し、集合中のタプルを新たなタプルで置き換え、集合の交わりを処理するステップを含むことを特徴とする請求項4記載の最適化方法。

【請求項6】前記ステップ(d)は下記のステップ(d6)を含むことを特徴とする請求項5記載の最適化方法：

- (d6) プログラム中のある点での2つの参照が一貫してメモリ中の同一のロケーションに関連するか、決して同一のロケーションに関連しないが、一時的に関連するか、繰り返しに関連するか、どの様に関連するかが不確定かを識別し、この識別結果に回答して、冗長なロードとストアを消去し、ロードとストアの順序を変更する。

【請求項7】前記ステップ(d)は下記のステップ(d7)～(d11)を含むことを特徴とする請求項6記載の最適化方法：

- (d7) 正規化された形態  $a_j + b$  の2つの添え字参照  $a_1J + b_1$ ,  $a_2J + b_2$  の入力を受け入れる：ここで、 $a$  と  $b$  はループ不変整数式であり、 $J$  はソフトウェア・ループの繰り返しの間に繰り返しその値が修正されるスカラ変数である；
- (d8)  $a_1 - a_2 = 0$  かつ  $b_1 - b_2 = 0$  か否かをテストし、そうである場合には前記2つの参照は一貫して依存関係にあるとする；
- (d9)  $a_1 - a_2 = 0$  かつ  $b_1 - b_2 \neq 0$  かあるいは  $a_1 - a_2 \neq 0$  かつ  $b_1 - b_2$  は  $a_1 - a_2$  で割り切れないか、あるいは  $a_1 - a_2 = 0$  かつ  $b_1 - b_2$  は  $a_1 - a_2$  で割り切れかつ  $(a_1 - a_2)(b_1 - b_2)$  は  $J$  の値の範囲外にあるか否かをテストし（ここで  $\#$  は不等号である）、もしそうであれば、前記2つの参照を独立であるとする；
- (d10) ループ中で  $J$  がその値を1回しか取らないか否かをテストし、もしそうであれば前記2つの参照を過渡的に依存関係にあるとする；
- (d11)  $a_1 - a_2 \neq 0$ 、かつ  $b_1 - b_2$  が  $a_1 - a_2$  で割り切れ、かつ  $(b_1 - b_2)/(a_1 - a_2)$  が  $J$  の値の範囲外にあり、かつ  $J$  がその値を複数回取るか否かをテストし、もしそうであれば前記2つの参照を繰り返し依存関係にあるとする。

【請求項8】プロセッサに、より少ない参照を行うようにさせ、更に参照を並列に行わせるようにするステップを含むことを特徴とする請求項7記載の最適化方法。

【請求項9】拡張仮想レジスタ・エレメントの各々は中間表現中の添字付き配列変数のテキスト名に結合され、前記結合されたエレメントとテキスト名は複数のマップ・タプルの1つを構成することを特徴とする請求項1記載の最適化方法。

- 【請求項10】前記マップ・タプルは少なくとも1つ

のマップ・タプル集合の要素であることを特徴とする請求項9記載の最適化方法。

【請求項11】下記の(a)ないし(c)を設けてなる命令レベル・パラレル・プロセッサ用コンパイラ:

(a) メモリ中に複数の拡張仮想レジスタ・データ構造を生成する手段: 前記拡張仮想レジスタ・データ構造の各々は線形に順序付けされた複数の仮想レジスタ・エレメントを有する;

(b) テキスト名と実メモリ・ロケーションをメモリ中の前記拡張仮想レジスタ・エレメントに対応つける手段;

(c) 複数のテキスト名を前記拡張仮想レジスタ・エレメントの1つに対応つける手段。

【請求項12】前記拡張仮想レジスタ・データ構造をスカラ変数としてアクセスする手段を設け、前記拡張仮想レジスタ・エレメントが個別にアクセスできることを特徴とする請求項11記載のコンパイラ。

【請求項13】前記拡張仮想レジスタ・データ構造は関数 remap() に応答し、前記関数 remap() は以前には前記拡張仮想レジスタ・エレメントのうちの第1の名前を使ってアクセスできたものを第2の名前を使ってアクセスできるようにすることを特徴とする請求項12記載のコンパイラ。

【請求項14】下記のステップ(a)を有する命令レベル・パラレル・プロセッサ用コンパイラの間接表現出力を最適化する方法:

(a) ソース・プログラムの間接表現を動的単一代入中間表現に変換する: 前記動的単一代入中間表現では与えられた前記中間表現中の複数の代入ステートメントが拡張仮想レジスタ・データ構造を参照する複数の縮約化されたステートメントに変換されている。

【請求項15】前記ステップ(a)は、動的単一代入形態への変換のステップ中に、前記中間表現からコピー操作を削除するサブステップを有することを特徴とする請求項14記載の最適化方法。

【請求項16】重複したレジスタ代入参照を拡張仮想レジスタ中の複数のレジスタ・エレメントのうちの1つへの参照で置換するステップを有することを特徴とする請求項15記載の最適化方法。

【請求項17】以下のサブステップ(b)~(f)を有することを特徴とする請求項16記載の最適化方法:

(b) 標準化形態  $aJ + b$  で表したとき  $a_1J + b_1$ ,  $a_2J + b_2$  となる2つの入力添字参照を受け付ける: ここで  $a$  と  $b$  はループ不変整数式であり、 $J$  はソフトウェア・ループの繰り返し中にその値が繰り返し変更されるスカラ変数である;

(c)  $(a_1 - a_2 = 0)$  かつ  $(b_1 - b_2 = 0)$  が成立するかどうかを試験し、これが成立した場合は、これらの参照を「一貫して依存関係にある」とする;

(d)  $((a_1 - a_2 = 0)$  かつ  $(b_1 - b_2 \neq 0))$  または

$((a_1 - a_2 \neq 0)$  かつ  $((b_1 - b_2)$  が  $(a_1 - a_2)$  で割り切れる) または  $((a_1 - a_2 \neq 0)$  かつ  $((b_1 - b_2)$  が  $(a_1 - a_2)$  で割り切れない) かつ  $((b_1 - b_2)(a_1 - a_2)$  が  $J$  の値の範囲外にある) が成立するかどうかを試験し、これが成立した場合は、これらの参照を「独立である」とする;

(e)  $J$  がループ中にその値を1回だけ取るかどうかを試験し、それが成立した場合はこれらの参照を「過渡的に依存関係にある」とする;

10 (f)  $((a_1 - a_2 \neq 0)$  かつ  $((b_1 - b_2)$  が  $(a_1 - a_2)$  で割り切れる) かつ  $((b_1 - b_2)(a_1 - a_2)$  が  $J$  の値の範囲内にある) かつ ( $J$  がその値を複数回取る) が成立するかどうかを試験し、それが成立した場合は「繰り返し依存関係にある」とする。

【請求項18】プロセッサがより少ない回数しか参照を行わないようにし、かつ参照を並列に行うようにするステップを含むことを特徴とする請求項17記載の最適化方法。

20 【請求項19】拡張仮想レジスタ・エレメントの各々は前記中間表現中の添字付き配列変数のテキスト名と組み合わせられ、前記組み合わせられたレジスタ・エレメントとテキスト名は複数のマップ・タプルの1つを構成することを特徴とする請求項18記載の最適化方法。

【請求項20】前記レジスタ・エレメントとテキスト名は複数のマップ・タプルの1つを構成し、複数のマップ・タプルは少なくとも1つのマップ・タプル集合の要素であることを特徴とする請求項19記載の最適化方法。

【発明の詳細な説明】

30 【0001】

【産業上の利用分野】本発明は、一般に電子計算機用のプログラミング言語コンパイラに関するものである。特に本発明は、命令レベル・パラレル・プロセッサ用コンパイラにおいて入力ソース・コードが添字付き配列変数への複数参照を含む場合にコンパイラが生成するオブジェクト・コードを最適化するための方法に関するものである。

【0002】

40 【従来技術及びその問題点】最初の頃のコンピュータは、プロセッサに対して1つの命令ストリームと1つのデータ・ストリームを設けるいわゆるフォン・ノイマン・プロセッサ・アーキテクチャを使用した。このプロセッサは、各命令を順次実行し、その際1つの記憶領域のデータに対して処理動作を行う。このような古典的プロセッサは、単一命令単一データ(SISD)プロセッサとして知られている。プロセッサのクロック速度が速くなるに従って、SISDアーキテクチャは高い処理スループットを達成することの障害となってきた。多くのコンピュータプログラムは、個別ではなく並列に実行することができ命令のグループまたはブロックを有する。

## 5

そのため、コンピュータの研究者達は、並列処理モデルのファミリーを含む他のアーキテクチャを開発するに至った。

【0003】A. 命令レベル・パラレル・プロセッサ  
パラレル・プロセッサの1つのタイプは命令レベル・パラレル (InstructionLevel Parallel, ILP) プロセッサとして知られている。ILPプロセッサにおいては、スケジューリングや同期化を行う計算の基本単位は、個々の加算 (add)、乗算 (multiply)、ロード (load) またはストア (store) 操作のようなプロセッサ命令である。相互依存しない命令は並列にロードされ実行される。

【0004】ILPプロセッサを用いると、命令スケジューリングあるいは同期化はプログラム実行の間に行う必要がなく、一部の決定はプログラムのコンパイル時に行うことができる。例えば、コンパイラが2つの操作が独立している (どちらも他方の操作の結果を入力として必要としない) ことを確認できるのであれば、それらの操作は並列に実行することができる。ILPプロセッサの動作を最適化する1つの方法は、実行時決定は全く不可能であるという仮定の下にコードを生成するコンパイラを作ることである。このコンパイラは、全てのスケジューリング及び同期化をおこなう。従って、プロセッサは実行時にはコードの順序変更 (re-ordering) をほとんど行わず、これによってリンク及び実行速度が改善される。

【0005】B. データフロー・プロセッサ  
もう1つのパラレル・プロセッサのタイプはデータフロープロセッサであり、このタイプは最大限に並列化されたプログラム実行が可能である。データフロー・プロセッサにおいては、命令は、必要データが全て処理に使える状態になるとすぐ実行される。データフロー・アーキテクチャ及びそのためのコンパイラは以下の参考文献に説明されている：

J. B. Dennisの"First Version of a Data Flow Procedure Language", Proceedings of Programming Symposium, Paris, 1974 (Lecture Notes in Computer Science 19 (Springer-Verlag, Berlin, 1974d)に再録)

K. R. Traub の"A compiler for the MIT Tagged-Token Dataflow Architecture", Technical Report MIT/LCS/TR-370 (MIT Laboratory for Computer Science, Cambridge, Mass., August, 1986)

【0006】C. ベクトル・プロセッサ

ベクトル・プロセッサも特定の形式の並列プログラムを実行することができる。当技術分野においては周知のように、ベクトル命令は、ループ本体中に単一の操作を有するループと等価である。例えば、ベクトル V のベクトル累算命令：

V := ++1;

は、以下の疑似コード・ループと等価である：

## 6

for Index := 1 to ベクトル長 do V[Index] := ++1

その結果、ベクトル・プロセッサ用コンパイラにおけるオブジェクト・コード生成は、主としてループ構造をベクトル命令に変換することに力を入れる。このようなベクトルコードの最適化は配列の添字の詳細な解析を必要とする。例えば、コンパイラは、2つの参照が同じ実メモリ・ロケーションに対するものであるかどうかを判断し、それらの参照を統合しなければならない。ベクトル・コンパイラの従来技術に関するまとめは、H. Zima 及び B. Chapman の "Supercompilers for Parallel and Vector Computers" (ACM Press, Association for Computing Machinery, New York, 1990) に記載されている。

【0007】残念なことには、これらの従来技術のベクトル・プロセッサ用コンパイラは、(冗長なあるいは部分的に冗長な配列参照の除去のような) 最適化より、むしろ (ループ交換や分配 (loop interchange and distribution) のような) プログラム変換に焦点が絞られていた。例えば、コード・ブロック 1A の FORTRAN コード・ループは B(I-1) の冗長なロードを行っている。

○コード・ブロック 1A

DO 10 I = 2, 50

A(I) = B(I) + B(I-1)

10 CONTINUE

ベクトル・コンパイラは、ベクトル B の要素の2つのロードが1インデックス・エレメントだけスキューされているということについては、典型的なベクトル・プロセッサのハードウェアでは利用することができないため、これを認識しようとはしない。しかしながら、当技術分野においては周知のように、コード・ブロック 1B の形式のコードを用いて冗長なロードが除去されるならば、ILPプロセッサはより効率的となる。

○コード・ブロック 1B

T = B(1)

DO 10 I = 2, 50

S = B(I)

A(I) = S + T

T = S

10 CONTINUE

【0008】コード・ブロック 1B を使用した場合には、ILPプロセッサは4番目と5番目のステートメントを同時に実行することができ、これによって1回のループ反復につきロード操作が1つ少なくなるので、性能が改善される。このように、ベクトル・プロセッサ最適化技術はILPプロセッサには適しない。

【0009】D. 従来技術の欠点

従来のSISDプロセッサでは、処理または問題解決における並行性の機会がほとんど得られない。従来の大部分のコンパイラはSISDプロセッサ用に書かれているので、パラレル・プロセッサのための最適化はなされて

## 7

いない。一方、従来の大部分のコンパイラは、任意の制御フロー（すなわちGOTO型の分岐命令及び構造化されていない繰り返し制御フロー）を持つプログラムを解析し、最適化することができる。

【0010】ILPプロセッサ及びコンパイラは、分岐命令が決定するところに従い、概ね逐次的に、1つの基本オブジェクト・コード・ブロックからべつの基本オブジェクト・コード・ブロックへ進む制御フローを与える。しかしながら、ILPコンパイラは、一部の命令を並列に動作させることによってスカラ性能を改善する。例えば、ILPコンパイラは、最も内側のループ・レベル内の有限個の連続した命令を並列実行させることによって、ループ構造を最適化することができる。この最適化のためには、コンパイラは、並列に実行される程度に接近した一連の命令の間の相互依存関係を解析するだけでよい。特に、2つの命令間の依存関係の性質は、この依存関係が侵犯される、つまり1つの演算からもう一方の演算へ結果がレジスタで持ち込まれる場合においてのみ重要である。

【0011】例えば、依存関係解析は、コード・ブロック2Aのように同じネスティング・レベルの複数の「兄弟関係」ループを有するコード・ブロックの場合と、コード・ブロック2Bに示すような依存関係を持つロードが入っているループでは異なる。

○コード・ブロック2A

```

DO 10 I = 1, N
X(I+K) = ... [ストア]
10 CONTINUE
.
.
.
DO 20 I = 1, N
... = X(I+L) [ロード]
20 CONTINUE
○コード・ブロック2B
DO 30 I = 1, N
10 ... = X(I-K) [ロード]
.
.
.
20 X(I) = ... [ストア]
30 CONTINUE

```

コード・ブロック2Aのプログラム・セグメントをコンパイルしているときは、ステートメント10と20の2つのループのストア・ステートメントとロード・ステートメントの間の依存関係は全く問題とならない。これら2つのループは基本ブロックであるため、ILPコンパイラはこれらのループの順序を守るので、これら2つのループが逆の順序で実行されるようスケジューリングする危険性はない。また、ベクトルXが小さいというこ

## 8

とが既知でない限り、CPUのレジスタは、第1のループにおけるロード操作によってそこにストアされた値を全て保持して第2のループにおけるロードを除去するには容量が通常不十分であるため、最適化はほぼ不可能である。

【0012】同様に、コード・ブロック2Bのロードとストアをコンパイルするとき、ロード操作とストア操作がコード中ではるかに離れている場合は、これらの操作が通常依存関係にあるかどうかはいつでもよい。操作間の距離がこのように大きいと、これらの操作が逆順にスケジューリングされることは全くない。従って、最大の関心事は、これらの操作が時間的に互いに近接しているかどうかである。

【0013】しかしながら、部分的な依存情報でも、ILPコンパイラの性能に役立つ。多数の反復によって隔離されている操作については証明が得られない場合でも、コンパイラが隣合った繰り返し中の操作が互いに独立していることを証明し得るならば、性能を著しく向上させることができる。このことは、ベクトル化コンパイラあるいは並列化コンパイラについては当てはまらない。

【0014】例えば、コード・ブロック2Bにおいて、ステートメント20のストアがステートメント10のロードに依存するものと仮定する。スカラ変数Kがゼロか負であると、繰り返しは不可能である。Kがゼロより大きいならば、DO 30のループの与所の繰り返しにおける行10のロードは、繰り返しの回数がK回前の行20のストアに依存する。Kが未知の場合は、コンパイラは最悪のケース(K=1)を想定しなければならず、隣り合った繰り返しのオーバーラップは全く不可能である。コンパイラがK>1であるということだけでも証明できれば、この2倍の性能を達成することができる。控え目に仮定して、K=2であり、隣り合った繰り返しの開始頻度をほぼ2倍にすることができる。

【0015】残念なことには、ILPアーキテクチャは従来の高水準コンピュータプログラミング言語コンパイラ及びコンパイラ最適化技法に充分適してはいない。当技術分野においては周知のように、ソース言語コンパイラは、最適化及びオブジェクト・コード生成前に中間表現(Intermediate Representation, IR)として知られている中間コードを作る。しかしながら、大部分の従来技術のIRは命令レベル並行処理を表現することができない。更に、従来技術のコンパイラは添字付き変数に対する参照の良好な最適化を行っていない。

【0016】

【目的】従来のコンパイラ及びオブティマイザ(optimizer)では、ILPプロセッサにとって最大限効率的なコードが生成されない。従って、プロセッサ及びコンパイラ的设计者は、添字付きの配列変数に対する参照を処理するための最大限に効率的なILPプロセッサコード

を生成するILPコンパイラ最適化方法があれば有用であろうということを理解できるだろう。

【0017】また、このような設計者は、従来のスカラ変数技法の力を配列への添字付き参照を含むあるクラスのプログラムに適用した最適化方法の有用性も理解できるであろう。

【0018】また、従来のコンパイラにより生成される中間表現はILPアーキテクチャに余り適してはいない。従来のオブティマイザは、構造化ループについては効果的であるが、条件付き分岐命令が埋め込まれているループのような構造化されていない繰り返し構造に適用すると劣悪なコードを生じる。従って、プロセッサやコンパイラの設計者は、任意の繰り返し制御フローを持つ最も内側のループで使用する最適化方法が有用であることを理解できよう。

【0019】また、プロセッサ及びコンパイラの設計者は、命令レベル並行処理のために最適化した依存関係解析を行うことが可能なコンパイラが得られるならば、これも評価するであろう。

【0020】データフロー・モデルはILPコンパイラに好適なIR生成のための基礎を与えてくれる。しかしながら、データフローIRは、従来の高水準言語プログラム中で与えられる任意の構造化されていない制御フロー・パスを扱うように拡張しなければならない。更に、データフロー・モデルは、並列的にデータを使用できるようにするために過大な数のコピー操作を必要とする。当業者であれば、最適化法を用いて不必要なコピー操作を除去するILP最適化方法が有用であろうということを理解できるであろう。

【0021】また、コンパイラやプロセッサの設計者は、ループを含んだ任意のフローグラフの存在下で添字付きのメモリ・ロケーション間の関係を含む操作間の依存関係を最適化し、また不必要なコピー操作のない最適化方法が有用であることを理解するであろう。また、主要な注意が添字付き参照に対して確実に払われるようにすることも好都合であろう。

#### 【0022】

【概要】従って、本発明は、ILPプロセッサ用のコンパイラ中間表現(IR)を最適化するためのプロセス、及びこのプロセスを実現するためのデータ構造を提供するものである。本発明のプロセスは、ランダム・アクセス・メモリのようなメモリ記憶手段へのアクセス及び磁気ディスク記憶装置のようなプログラム用大容量記憶手段へのアクセスを有する電子計算機またはデータ処理装置上で動作するコンパイラ・コンピュータ・プログラムとして実施することが望ましい。このコンパイラ・プログラムは、プログラム用大容量記憶手段に記憶された入力ソース・プログラムを読み込み、疑似機械命令を用いてメモリ中にソース・プログラムの動的単一代入中間表現(dynamic single assignment intermediate repre-

ntation)を生成する。動的単一代入中間表現を生成するために、コンパイルの間に、コンパイラは、ソース・プログラム中で定義されている変数を記憶するためにメモリ中に複数の仮想レジスタを生成する。このプロセスにより、コンパイラは、IRコードを文字通り読んだ場合には左辺に同じ仮想レジスタがある代入ステートメントが複数本ある場合であっても、同じ仮想レジスタに対しての動的実行パス上でも決して複数回の代入が行われることがないようにする。このプロセスは、無限の、直線状に配列された仮想レジスタ・エレメントの集合よりなるEVRデータ構造を用いて実現され、拡張仮想レジスタ(EVR)上でremap()関数が定義されている。EVRパラメータを付けてremap()関数を呼び出すと、リマップ操作前には[n]としてアクセス可能であったEVRエレメントがリマップ操作後は[n+1]としてアクセス可能となる。このプロセスにより、プロセッサは、動的な複数のマップ・タプル(map tuple)よりなる添字付き参照マップを生成して使用する。各マップ・タプルは、テキスト表現された名前(textual name)でアクセス可能な実メモリ・ロケーションをEVRエレメントと対応付ける。このようにして、コンパイラは、マップ・タプルを用いてテキスト表現された名前をEVRエレメントで置換することができ、これによって出力の中間表現から不必要なロード操作を除去することができる。

#### 【0023】

【実施例】以下の本発明の実施例の詳細な説明においては、説明をわかり易くするために一部特殊な用語を使用する。しかしながら、本発明は、ここで使用する特定の用語に限定されるものではなく、実質的に同様の結果を達成するために実質的に同様な動作をする技術的に均等な事柄全てを含むものとする。本願の技術分野の当業者であれば、コンパイラの理論と実践に関する教科書で論じられている全ての概念及び技術を理解できるであろう。そのような教科書の1つに、Addison-Wesley (Reading, Mass.)によって1985年に出版されたA. Aho, R. Sethi と J. Ullman 共著の“Compilers: Principles, Techniques and Tools”があり、その用語及び定義は本願明細書でも使用する。

【0024】本発明は、コンパイラの中間表現(IR)コードを最適化するための方法を提供すると共に、その方法を実施するためのデータ構造を提供するものである。好ましくは、図1に示すように、本発明の方法は、ランダム・アクセス・メモリのようなメモリ・ストレージ手段(メモリ)4へ、また磁気ディスク・ストレージのようなプログラム用大容量ストレージ手段(ディスク・ストレージ)6へアクセスできる電子計算機またはデータ処理装置(プロセッサ)2上で動作する従来のコンパイラ・コンピュータ・プログラム8を用いて実施する。まず始めに、コンパイラ8は、プログラム用大容量

ストレージ手段に記憶されたFORTRANプログラミング言語で書かれたプログラムのような入力ソース・プログラム10を読み込み、以下に説明する方法により疑似機械命令を用いてメモリ中にソース・プログラムの動的単一代入中間表現 (Dynamic Single Assignment Intermediate Representation, DSA IR) 12を生成する。

#### 【0025】A. 依存関係解析及び単一代入

ILPコンパイラが最大限の並列プログラム表現を達成するためには、正しい意味を確保するのに本質的ではない命令依存性を全て除去することが必要である。当技術分野においては周知であり、また H. Zima と B. Chapman 共著の "Supercompilers for Parallel and Vector Computer" (ACM Press, New York) に開示されているように、命令相互依存関係 (instruction inter-dependency) は、真の依存関係 (true dependency)、反依存関係 (anti-dependency) 及び出力依存関係 (output dependency) に分類することができる。正しい意味のためには真の依存関係のみが本質的である。反依存関係及び出力依存関係は、いくつかの操作の結果を同じ変数に対して代入する結果生じるものであり、本発明を使用する場合はこのような結果が現れることは決してない。

【0026】並列コンパイラの間中表現 (IR) においては、反依存関係及び出力依存関係は、コンパイラが結果を仮想レジスタに決して2回以上代入することがないようにすることにより除去することができる。この「単一代入 (single assignment)」の原理は、L. Tesler と H. Enea 共著の "A Language Design for Concurrent Processes", Proceedings AFIPS Sprint Joint Computer Conference (1968) 403~408ページに開示されている。単一代入は、上に引用した A. Aho 他の教科書に開示されている値番号付け手続 (value numbering procedure) を用いて単一の基本コード・ブロックで実現することができる。

【0027】当技術分野においては周知のように、コンパイラは、代入操作がIRに入れられる毎にメモリ4に別個の仮想レジスタ14を生成するつまり設定することによって、ソース・プログラムの静的単一代入IRを得ることができる。静的単一代入IRにおいては、各仮想レジスタはプログラム中で代入の左辺にちょうど1回だけ現れる。しかしながら、このような代入がループ構造中に起こった場合には、ループを何回も反復すると、同じ仮想レジスタに対して何回も代入が行われる。そのような複数回の代入は動的単一代入の原理に違反し、ループ中の操作間に反依存関係を生じさせる。更に、特定の仮想レジスタの多重定義を区別することは不可能であり、そのために依存関係を精密に指定することは不可能になる。従って、当業者であれば、動的な意味で単一代入を可能にすることによってこれらの問題を解消するIRを高く評価するであろう。

#### 【0028】B. 動的単一代入

並行性を強化し、ループ構造内で静的単一代入を用いることの問題を解消するための1つの方法は、IRを動的単一代入 (DSA) 形式にすることによるものである。

【0029】図3に示すように、コンパイラ8でのIR最適化には、ブロック60で始まるプロセスを使用する。本発明の1つの特徴は、本願中において動的単一代入 (DSA) と定義し、図4に詳細に示すIR最適化方法のステップ62である。IRをDSA法によって作成すると、文字通りに読んだ場合には、IRコード中に左辺に同じ仮想レジスタがある複数の代入ステートメントがある場合でも、同じ仮想レジスタにはどの動的実行パスについても決して2回以上代入されることはない。この状況は静的単一代入とちょうど正反対である。静的単一代入では、ひとつの仮想レジスタはステートメントの左辺に2回以上現れることは決してないが、ループ内ではその仮想レジスタに対して必然的に何回も代入が行われる。

【0030】本発明においては、入力ソース・プログラム10は、3つの基礎的なツール、すなわち拡張仮想レジスタ (EVR) データ構造、remap() 関数、及び複数のマップ・タプルよりなるメモリ・マップ、を用いて動的単一代入IR12に変換される。これらのツールは、図3のステップ66、67及び68を実行するために用いられる。

【0031】図2(A)に示すように、EVRデータ構造36は、好ましくは、無限の直線状に配列された仮想レジスタ・エレメント38、40、42の集合よりなり、特別の操作30、すなわち remap() がその上で定義されている。remap() 関数の唯一のパラメータはEVRの名前である。EVRに t という名前が付いていれば、EVRのエレメントは t[n] でアドレス指定し、読み出し、書き込むことができる。ただし、nは任意の整数である。ここでは、便宜上最初のエレメント t[0] が t とも呼ばれる。remap(t) 関数を呼び出すと、remap(t) 操作の前に t[n] としてアクセス可能なエレメントがあった場合、そのエレメントは remap(t) 操作の後では t[n+1] としてアクセス可能となる。ここで、ラベル t は下記のコード・ブロック3B~3Fの仮想レジスタ名に対応付けるために意図的に用いたものである。

【0032】実際、remap() 関数は、拡張仮想レジスタの1つのエレメントを別のエレメントのEVR名を用いてアクセスできるようにする。EVR t への結果の連続した代入の間に remap(t) が実行されると、t に代入された前回の値は t[1] に入っており、今度実行されるときは別のエレメント t[0] に対して行われる。図2(B)は、この操作を、IRコード・セグメント50と、参照番号52が付けられたエレメント t[0] の内容と、参照番号54のエレメント t[1] の内容を並置することで示している。オブティマイザが命令 [t=2] に

13

遭遇すると、値[2]をエレメント t[0] にストアする式がオブティマイザIR出力に付加される。次に、remap(t)関数がIR出力に付加される。remap()関数は、矢印53で示すように、t[0]の内容をt[1]へ移動させる。次に、命令[t=6]によって値[6]がt[0]にストアされる。プログラム・テキスト上は、結果がt[0]に繰り返し代入されているように見えるが、その時々異なるEVRエレメント及び実メモリ・ロケーションへストアすることによって、動的に単一代入ルールが守られている。

【0033】remap()関数は、図3のブロック64に示すように、ソース・プログラムの最適化及びIRへの変換の間にDSA IR12に付加される。IR中での各代入の前に1つのremap()ステートメントが付加される。

【0034】EVR及びremap()関数を使用して、ソフトウェア・ループで動的単一代入を達成するためのプログラムがコード・ブロック3A~3F及び図4に示されている。

【0035】コード・ブロック3Aは、FORTRAN言語による簡単なループを示す。例として、コード・ブロック3Aは、本発明を組み込んだコンパイラ8に入力されるソース・プログラム10の一部を形成することができる。

【0036】コード・ブロック3Bは、ステップ72で作成されるコード・ブロック3Aのループに対応する逐次的IRである。

【0037】コード・ブロック3Cは、コード・ブロック3A及びステップ74の単一代入IRである。

【0038】コード・ブロック3Dは、ステップ76に示すように、ループ本体内の各代入の前にremap()操作を挿入することによって得られる、コード・ブロック3Bの逐次的IRについての正準動的単一代入IRである。

【0039】コード・ブロック3Eは、全てのremap()操作をループの始めへ移動させた(ステップ78)後の、DSA形式のコード・ブロック3Dを示している。

【0040】コード・ブロック3Fは、コピー最適化を実行した(ステップ79)後のコード・ブロック3Eを示す。

【0041】

10

20

30

40

14

#### コード・ブロック3A コード・ブロック3B (FORTRANループ) (逐次的IR)

```

      % t00 = 0, t01 = 1
      % t02 = 5, t03 = 50
      K = 0
      J = 0
      DO 10 I=1, 50
        L = J
        J = J + K
        K = L
      10 CONTINUE

      K = J + 5

```

#### コード・ブロック3C (静的単一代入IR)

```

      % t00 = 0, t01 = 1
      % t02 = 5, t03 = 50
      s0 t04 = copy(t00)
      s1 t05 = copy(t01)
      s2 t06 = copy(t01)
      s10 t10 = 0 (t04, t14)
      s11 t11 = 0 (t05, t13)
      s12 t12 = 0 (t06, t15)
      s3 t07 = copy(t11)
      s4 t13 = ladd(t11, t10)
      s5 t14 = copy(t07)
      s6 t15 = ladd(t12, t01)
      s7 t08 = ile(t15, t03)
      s8 brt(t08, s10)
      s9 t9 = ladd(t13, t02)

```



```

15      コード・ブロック 3 D
      (正準動的単一代入)
      % t0 = 0, t1 = 1
      % t2 = 5, t3 = 50
      K = 0
      J = 0
      DO 10 I=1, 50
          L = J
          J = J + K
          K = L
      10 CONTINUE
          K = J + 5
      15      コード・ブロック 3 E
      (remap()の移動後)
      % t0 = 0, t1 = 1
      % t2 = 5, t3 = 50
      s0 t4 = copy(t0)
      s1 t5 = copy(t1)
      s2 t6 = copy(t1)
      s10 remap(t7)
      s11 remap(t5)
      s12 remap(t4)
      s13 remap(t6)
      s14 remap(t8)
      s3 t7 = copy(t5[1])
      s4 t5 = iadd(t5[1], t4[1])
      s5 t4 = copy(t7)
      s6 t6 = iadd(t6[1], t1)
      s7 t8 = ile(t6, t3)
      s8 brt (t8, s10)
      s9 t9 = iadd(t5, t2)

```

10

20

30

40

50

```

16      コード・ブロック 3 F
      (コピー最適化後)
      % t0 = 0, t1 = 1
      % t2 = 5, t3 = 50
      s0 t4 = copy(t0)
      s1 t5 = copy(t1)
      s2 t6 = copy(t1)
      s11 remap(t5)
      s13 remap(t6)
      s14 remap(t8)
      s4 t5 = iadd(t5[1], t5[2])
      s6 t6 = iadd(t6[1], t[1])
      s7 t8 = ile(t6, t3)
      s8 brt (t8, s10)
      s9 iadd(t5, t2)

```

【0042】最初の5行の初期化ステップを除いては、コード・ブロック3B～3Fの各行は、コード・ブロック3Aのソース・コードの行と対応する。全ての操作のソース及びデスティネーションは、メモリ4にストアされ、t1～t10のラベルが付与された仮想レジスタ14である。当技術分野においては周知のように、上記コード・ブロックの“iadd”命令は、CPUの整数加算操作を表し、“ile”命令は整数型の以下か否かの比較であり、“brt”命令は、「真ならば分岐せよ (branch if true)」の操作である。コード・ブロック3Aのループにより、IRは、番号をつけられたステートメントへの分岐参照ができるようにするためにステートメント番号s0～s14を用いる必要がある。

【0043】コード・ブロック3Bの逐次的中間表現はステップ72で生成され、同じ仮想レジスタへの代入の場合がいくつか含まれている。例えば、仮想レジスタt04とt06に対してはブロック3A中では代入が2度行われる。レジスタt4はステートメントs0及びs5で代入があり、レジスタt5はステートメントs1及びs4で代入がなされ、レジスタt6はステートメントs2及びs6で参照される。前に述べたように、これは反依存関係を持ち込み、並行性を妨げるので好ましくない。

【0044】同じ仮想レジスタへの反復代入は、ステップ74で作成されるコード・ブロック3Cの静的単一代入IRを用いて排除することができる。しかしながら、より多くの仮想レジスタが必要であり、コード・ブロック3Bと3Cには同数のロード操作がある。

【0045】反依存関係及び反復代入は、どちらもステップ76で生成されるコード・ブロック3Dの正準DSA形式を用いて取り除くことができる。この形式への変換は、s0からs2までのステートメントの左辺のt4、t5、t6への参照をt4[1]、t5[1]、t6[1]への参照で置換することにより行われる。コード・ブロック3DのIRには、仮想レジスタへの各代入の前にremap()

操作が挿入される。ステートメントs4でt5を使用す

る前に t5 に対する remap() 操作を挿入するので、s4 の右辺の t5 への参照を t5[1] に変えることが必要となる。同様に、ステートメント s6 は t6[1] を使うように変更される。しかしながら、コード・ブロック 3D は、t4、t5 及び t6 が各々ステートメントの左辺に2度ずつ現れるので、まだ静的単一代入ルールに違反している。

【0046】従って、ステップ78に示すように、好ましくは、全ての remap() 操作をループ本体の先頭に移動させることにより更に形式的変換を行い、コード・ブロック 3E の IR を得る。t4 及び t5 に対する remap() をステートメント s4 及び s3 の前に動かすことにより、右辺の参照は t4[1] 及び t5[1] に変わる。

【0047】次に、意味のないコピー操作を取り除くことによって、当技術分野において周知の方法によりコピー最適化をステップ79において行うことができる。コード・ブロック 3E は、t7 が常に t5[1] と同じ値を有し、t4[1] が常に t7[1] と同じ値を有し、他方 t7[1] は t5[2] と同じ値を有するというに着目することによって、コピー最適化されたコード・ブロック 3F に変換することができる。t7 及び t4 を各々 t5[1] 及び t5[2] で置換した後は、コード・ブロック 3E のステートメント s3 及び s5 はデッド・コードであり、コード・ブロック 3F においては省略してもよい。その結果、ステートメント s10 及び s12 は不必要である。コード・ブロック 3E のステートメント s4 には、今回の繰返しに関する加算の結果が前の2回の繰返しに関する同じステートメントの結果の和であるということが実効的に記述されている。この表記法によれば、コード・ブロック 3B をより短く、より効率的な方法で表すことが可能である。

【0048】C. 添字付き配列変数 (subscripted array variable) の解析

本発明によれば、添字付きの配列変数の参照は、コンパイラにスカラ変数参照の場合と全く同様に配列参照を扱わせることによって最適化することができる。当技術分野においては周知のように、 $X(I, J)$  のような配列の要素は、同じ配列要素のメモリ・ロケーションを、例えば  $X(I, J)$ 、 $X(I+1, J-1)$  あるいは  $X(L, M)$  等のように種々のやり方で参照することができるという点を除けば、スカラ変数と何ら異ならない。I, J, L, M の値が適切であれば、これら3通りの参照はいずれも同じメモリ・ロケーションを指示する。

【0049】図4のステップを使って図3のステップ62でDSA形式を確立した後、コンパイラの配列参照の最適化は、図3のステップ67に示すような依存性解析によって行うことができる。依存性解析は、図6及び図7のデータフロー解析ステップを用いて行われる。重要なステップは、図6でテストされるように、コンパイラに2つの参照が同じメモリ・ロケーションを指している

ということを認識させることである。プログラム中の、 $I = I+1$  の形式の累算ステートメントの直前のある点に  $X(I)$  があるということをコンパイラが知れば、 $X(I-1)$  はその累算ステートメントの後では、累算ステートメントより前の  $X(I)$  と同じメモリ・ロケーションを指す。

【0050】好ましくは、同じ配列に対する全ての参照の添字は、 $aJ+b$  の形式となるように、そこまでのコンパイラ・ステップで正規化されているものと仮定する。ここで、a, b はループ不変式であり、J はループ内で繰返し修正される何らかの整数値を取るスカラ変数である。 $aJ+b$  の形式の添字では、J は添字変数と呼ばれる (J は一般にはループ・インデクスである)。同じ配列に対する全ての参照の添字は同じ変数 J を用いて表すことが好ましい。

【0051】ループ内では、J のような添字変数の定義は全て  $J = J+K_i$  の形式とすることが好ましい。ここで、これらの  $K_i$  はループ不変整数量である。ループ内にはこのような添字変数の定義が複数あってもよい。

【0052】添字付き参照の依存性解析

本発明の配列最適化方法には、ブロック67 (図3) 及び図5に示すように、添字付き参照の依存性解析を行うことが含まれる。

【0053】まず、別名で行われる (aliased) 可能性のある参照の集合が作られる (図5、ブロック82)。ブロック86においてテストされるように、同じ配列に対してなされるかまたはその可能性がある参照は、全て1つの集合に入れられる。一度には1つのエイリアス集合しかテストされないのので、このコードについてのパスが何回か実行される。このプロセスは、ステップ82～94のループに示されている。

【0054】この方法では、図5のデータフロー解析ステップ91において、2つの配列参照が同じメモリ・ロケーションを指示するものであるかどうかをテストする。この手順を図6に詳細に示す。2つの添字  $a_1J + b_1$  及び  $a_2J - b_2$  がプログラムの同じ部分にあれば、同じメモリ・ロケーションを指示するためには、 $(a_1 - a_2)J + (b_1 - b_2)$  が、J がある整数値を取ったときに0にならなければならない。この方法では、2つの添字付き参照  $X(a_1J - b_1)$  と  $X(a_2J - b_2)$  が同じ近隣の場所にあると確認したならば、次にこれら2つが同じメモリ・ロケーションを指すものであるかどうかをテストする。これについては、図6のステップ204～226に示し、以下に考察するように、5通りの結果が有り得る。(ここでは、一次元配列についてのみ考察するが、本発明の方法は、各次元毎に添字を別個に考えるかあるいは線形化した配列の単一の添字について考察することにより、多次元配列にも適用することが可能である。)

【0055】1. 常に違うロケーションを参照する

ステップ204、206、210、212及び214に

示すように、少なくとも次の3つの場合には、2つの参照は同じメモリ・ロケーションを指さない：

- (1) (ステップ204, 206)  $(a_1 - a_2) = 0$  だが  $(b_1 - b_2) \neq 0$  (ここで  $\neq$  は不等号である。以下同様)
- (2) (ステップ204, 210)  $(a_1 - a_2) \neq 0$  であり  $(b_1 - b_2)$  は  $(a_1 - a_2)$  で割り切れない
- (3) (ステップ204, 210, 214)  $(a_1 - a_2) \neq 0$  であり  $(b_1 - b_2)$  は  $(a_1 - a_2)$  で割り切れるが、 $(b_1 - b_2)(a_1 - a_2)$  は J の値が取る範囲外にあると証明することができる。

【0056】2. 常に同じロケーションを参照する  $(a_1 - a_2) = 0$  かつ  $(b_1 - b_2)$  (ステップ204~208) ならば、2つの参照は J の値にかかわらず同じメモリ・ロケーションに対するものである。従って、それら2つの参照はテキスト上は同じである。

【0057】3. 繰り返し同じロケーションを参照する  $(a_1 - a_2) \neq 0$  で  $(b_1 - b_2)$  が  $(a_1 - a_2)$  で割り切れ、かつ  $(b_1 - b_2)(a_1 - a_2)$  が J の値が取る範囲内にあることが証明できれば、2つの参照が同じメモリ・ロケーションに対してなされる J の値が1つある。更に、J は、ステップ204, 210, 214, 216, 220に示すように、この値を繰り返して取る。

【0058】4. 過渡的に同じロケーションを参照する  $(a_1 - a_2) \neq 0$  かつ  $(b_1 - b_2)$  が  $(a_1 - a_2)$  で割り切れかつ  $(b_1 - b_2)(a_1 - a_2)$  が J の値が取る範囲内にあると証明できれば、2つの参照が同じメモリ・ロケーションに対してなされるようになる J の値が1つある。しかし、ステップ214及び218に示すように、J はその値を1度だけ取る。J がループ・インデクスであるならば、これは、予期された結果である。

【0059】5. 不確定

時には、この方法で参照が上記の4つのカテゴリのどれかに入ることを証明することができない場合がある。例えば、ステップ200及び202に示すように、不確定の結果は、2つの参照がエイリアス同士であるかも知れない配列に対するものであったり、2つの参照が同じ配列に対するものであるが添字変数が異なっていたり、あるいは2つの添字の差が正規化された線形の形式になっていない場合に生じる。

【0060】本発明の方法においては、2つのメモリ・ロケーションは、上記1. (常に違うロケーションを参照する) の場合は「異なる」と見なされる (ステップ224)。また、2. (常に同じロケーションを参照する) の場合は「同じである」と見なされる (ステップ226)。その他の場合は「同じかもしれない」と見なされる (ステップ222)。別々のメモリ・ロケーションに対する複数の参照は独立しており、並列実行が可能である。一貫して同じメモリ・ロケーションに対する複数の参照は常に依存関係にあり、並行性を妨げるが、ロー

ド及びストアについての最適化の機会が得られることがしばしばある。残り3つの場合は、結果がいつも予測可能ではないので検討する価値はほとんどなく、「依存関係を有かもしれない」と分類される。

#### 【0061】データフロー解析

本発明による最適化は、次に図7に示すようなデータフロー解析へ移る。当技術分野においては周知のように、データフロー解析は、個々の基本コード・ブロック間のプログラムフローに対応しこのプログラムフローを表したデータフロー方程式を構築することによって行われる。ILPコンパイラでの添字付き参照の最適化を効率的にするために、本発明では、更にコンパイラ・プロセッサ・メモリにコンパイラ・マップを設ける。コンパイラ・マップは、プログラムの任意の点におけるメモリ・ロケーションと拡張仮想レジスタとの間の関係を記述する。好ましくは、このマップは  $\langle X(f(I)), y[n] \rangle$  の形式のマップ・タプルの集合よりなるデータ構造にストアする。マップ・タプルの2つのエレメントには各々 M-名及び R-名がラベルされる。マップ・タプルは、プログラムのその特定の点でテキスト表現された名前  $X(f(I))$  によってアドレス可能であるメモリ・ロケーションが、その同じ特定の点におけるその時の名称が  $t[n]$  である拡張仮想レジスタのエレメントと対応しているということを示す。マップ・タプルはメモリ中では整数対によって表すことができる。

【0062】ここで、S はその中のすべてのタプルが「メモリ・ロケーション  $X(f(I))$  の内容は、現在のところ、対応する EVR エレメント  $t[n]$  に入っている」というような同じ性質を有するようなマップ・タプルの集合であるとする。このような集合は従来のデータフロー・アルゴリズムによって扱うことも可能である。しかしながら、ステップ240で IR ステートメントを調べた後、本発明の方法においては、図7に示す5つの特別なデータフロー解析ステップを用いる。

【0063】1. マップ・タプルの挿入及び削除 (ステップ242~243)

M-名  $X(f(J))$  と同じかまたは同じである可能性があるメモリ・ロケーションへのストアが行われるプログラム中の各点で、マップ・タプル  $\langle X(f(J)), t[n] \rangle$  を S から削除する。ロード操作が  $X(f(J))$  を  $t[n]$  にロードするか、あるいはストア操作が  $t[n]$  を M-名  $X(f(J))$  へストアするプログラム中の各点で、マップ・タプル  $\langle X(f(J)), t[n] \rangle$  を集合 S に挿入する。

【0064】2. メモリ・ロケーションの再命名  
コンパイラがソース・コードステートメントを IR の表現式に翻訳する際、集合 S は前方に伝播される。コンパイラが  $J = g(J)$  の形式のステートメントに遭遇すると (ステップ244)、ステップ246に示すように、M-名が J の関数である全てのマップ・タプル  $\langle X(f(J)), t[n] \rangle$  が削除される。関数  $g(.)$  が許容的であり

(admissible) (ステップ248)、前方フロー解析が行われていると(ステップ249)、マップ・タプル  $\langle X(f(g^{-1}(J))), t[n] \rangle$  が集合に付加される(ステップ250)。関数  $g(\cdot)$  は、それが何らかのループ不変量によるインクリメント操作またはデクリメント操作に対応する場合にのみ許容的であると定義される。例えば、 $X(J+5)$  がステートメント  $J = J+2$  の前にアクセス可能であれば、 $X(J+3)$  はそのステートメントの後にアクセス可能であり、従って  $X(J)$  は許容可能である。同様に、集合  $S$  が  $J = g(J)$  の形式のステートメントを越えて後方へ伝播する場合は必ず、そのM-名が  $J$  の関数である全てのマップ・タプル  $\langle X(f(J)), t[n] \rangle$  が削除され、また関数  $g(\cdot)$  が許容的であれば、マップ・タプル  $\langle X(f(g(J))), t[n] \rangle$  が集合に付加される。

【0065】3. 拡張仮想レジスタ・エレメントの再命名

集合  $S$  が  $\text{remap}(t)$  の形式のステートメントを越えて前方に伝播すると、ステップ258に示すように、R-名が  $t$  のエレメントである全てのマップ・タプル  $\langle X(F(J)), t[n] \rangle$  がマップ・タプル  $\langle X(F(J)), t[n+1] \rangle$  で置換される。同様に、ステップ256に示すように、集合  $S$  が  $\text{remap}(t)$  の形式のステートメントを越えて後方へ伝播する場合は必ず、R-名が  $t$  のエレメントである全てのマップ・タプル  $\langle X(f(J)), t[n] \rangle$  が  $\langle X(f(J)), t[n-1] \rangle$  で置換される。

【0066】4. 合流操作 (meet operation) (ステップ260)

プログラム中で  $n$  本の制御フローパスが出会う点において、これらのパスに沿って伝播する集合を  $S_i$  ( $i = 1, \dots, n$ ) で表す。これらのパスの合流点をちょうど越えた点での集合  $S$  は次のようにして構成される。合流操作が論理積型の場合は、全ての入来パス  $i$  上の集合  $S_i$  中に  $\langle X(f(J)), t_i \rangle$  の形式のマップ・タプルがある場合にのみ、M-名  $X(f(J))$  を有するマップ・タプルが  $S$  に入れられる。言い換えると、M-名が論理積型合流を通して伝播するためには、同じM-名が各パスから到達しなければならない。更に、これが成立するとき、マッチングする全ての入来マップ・タプルの  $t_i$  がEVR名称においてもインデックスにおいても同じである場合は、同じR-名が出力マップ・タプルに入れられる。そうでない場合は、新しいEVR  $r$  がメモリ中で割り当てられ、 $r[0]$  が出力マップ・タプルのM-名として用いられる。合流操作が論理和型の場合は、いずれかの入来パスから入ってきた全ての入来マップ・タプルが出力集合  $S$  に入れられる。

【0067】これらの操作を用いて、プログラムの1つの点からのマップ・タプルは、プログラム中の同じ点で2つの配列添字の比較を行うことができるように、プログラム中の他の点に正しく伝播される。

【0068】その後、この方法では、問題とされている

全ての点において集合  $S_i$  の値を、安定した解が見つかるまで繰返し計算することにより、従来のデータフロー解析を実行する。プログラム中の任意の点での集合  $S$  は、どのM-名がEVRでアクセス可能であり、どの特定のEVRエレメント中にそれらのメモリ・ロケーションの内容が見出されるかを正確に指示する。

【0069】ここで、フロー方程式を解く反復プロセスが安定状態に収束するかどうかという疑問を抱くかも知れない。 $I = g(I)$  の形式のステートメントに遭遇する都度行われるM-名の再命名があるとすると、集合  $S$  は、ループの逐次反復の間に伝播するに従い際限なく成長であろう。5つのデータフロー解析ステップの後、ステップ262でウィンドウ化 (windowing) 技法を適用することによりメモリ・オーパフローを防ぐことができる。

【0070】例えば添字変数が単調に変化する場合、すなわち  $I$  がインクリメントするだけか、あるいはデクリメントするだけの場合は、集合  $S$  の際限のない成長は添字に対して最大値及び最小値を限定することにより抑止することができる。再命名の後、M-名がこれらの値のウィンドウの外に出た際には、そのマップ・タプルはマップ・タプルの集合から必ず放棄される。コンパイラが解析中のコードの領域内の全てのテキスト参照を含むようにウィンドウを設定すれば、有用な情報は失われない。M-名がいったんウィンドウの外に出て行ってしまうと、それ以後のループを回っての伝播はそのM-名をウィンドウの更に外側へ移動させるだけであり、プログラム中のどの参照とも同じになることは二度とない。従って、このM-名は値を持たず、放棄することができる。このように、ウィンドウを設定すると、マップ・タプル集合の「切捨て」が可能となり、反復プロセスを解へ向けて収束させることができる。

【0071】上述のようにウィンドウを設定することができない場合は、その代わりに、マップ・タプル集合を一定回数  $N$  だけループ回を回す。これによって、最大  $N$  回だけ離れた反復同士の間での依存関係に関する正確な情報を含むマップ・タプル集合が得られる。この回数を越えた先については情報は全く得られず、最悪の場合は、ある反復における全ての参照が反復回数  $N$  回以上離れた全ての参照に依存すると考えることも可能である。実際の使用に関しては、本願中で開示した方法は最適化されていないコードにも適用することができ、その場合各操作毎にメモリからソース・オペランドをロードし、結果を逆にメモリに保管して、メモリ・ロードが最適化されたDSA形式へのコードの変換を行わせる。

【0072】これを行うためには、上に説明したデータフロー解析の間に、 $S$  を既知のEVR中でアクセス可能であるM-名をリストするマップ・タプルの集合と定義する。次に、冗長なロードをIRロード中のテキスト表現された名前と既知のEVRエレメントを代入するこ

とによって除去する。その近隣についてのマップ・タプル集合中に対応するマップ・タプル  $\langle v, v \rangle$  を有する  $M$ -名  $v$  を指定するロード操作を除去することができる。それは、そのメモリ・ロケーションの内容は既知のEVRエレメントにあるからである。そのロード操作の結果を使用することになっている操作は、EVRエレメント  $v$  を参照するよう変更される。

【0073】冗長なストア操作を除去するためには、等価な後退フロー解析を用いることができる。例えば、プログラム中で  $v$  に「等しいかもしれない」変数に対するストアをおこなう点において、 $M$ -名として  $v$  を有する全てのマップ・タプルは、プログラム中のその点における集合  $S$  から削除される。

【0074】データフロー解析は、上述のように、マップ・タプルを集合  $S$  に対して加除しつつ続けられる。全てのステートメントの処理が終わると、プログラム中の任意の点における集合  $S$  によって、どの  $M$ -名がEVR中でアクセス可能であるか、そしてそれらのメモリ・ロケーションの内容がどの特定のEVRエレメント中にあるかが正確に指示される。

【0075】これ以外の本発明の説明は、「命令レベル並行処理のためのデータフロー及び依存性解析 (Data Flow and Dependence Analysis for Instruction-Level Parallelism)」という名称の論文に記載されている。また、本発明は、本願に特定の開示した以外の形で実施することができる。従って、本発明の範囲は特許請求の範囲の記載のみによって限定されるものである。

【0076】

【効果】以上詳細に説明したように、本発明によれば、命令レベル・パラレル・プロセッサのロードの最適化を行うことができる。

【図面の簡単な説明】

【図1】コンパイラとプロセッサのブロック図。

【図2】拡張仮想レジスタとremap関数を使用する前と、後で仮想拡張コンパイラにストアされている値を説明する図。

【図3】最適化方法のフローチャート。

10 【図4】図3のフローチャートの部分ステップとなる動的単一代入変換方法のフローチャート。

【図5】図3のフローチャートの部分ステップとなる依存性解析と最適化のフローチャート。

【図6】図3のフローチャートの部分ステップとなるメモリロケーション一貫性テストのフローチャート。

【図7】図3のフローチャートの部分ステップとなるデータフロー解析方法のフローチャート。

【符号の説明】

2 : プロセッサ

20 4 : メモリ

6 : ディスク・ストレージ

8 : コンパイラ

10 : ソース・プログラム

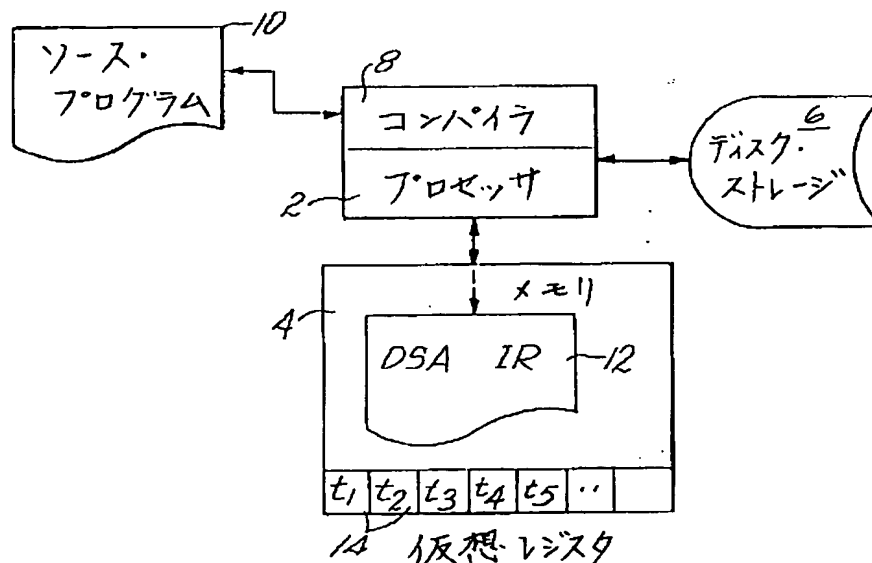
12 : DSA IR

14 : 仮想レジスタ

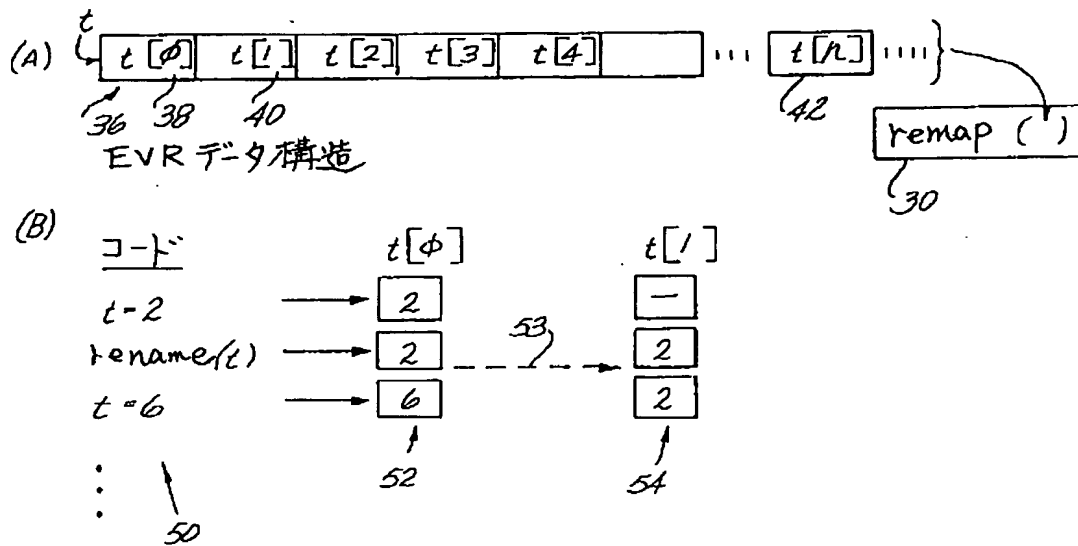
36 : EVRデータ構造

38、40、42 : 仮想レジスタ・エレメント

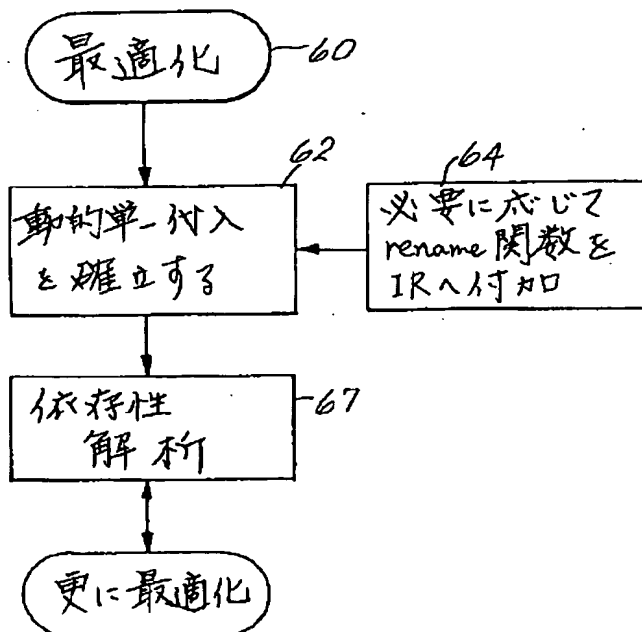
【図1】



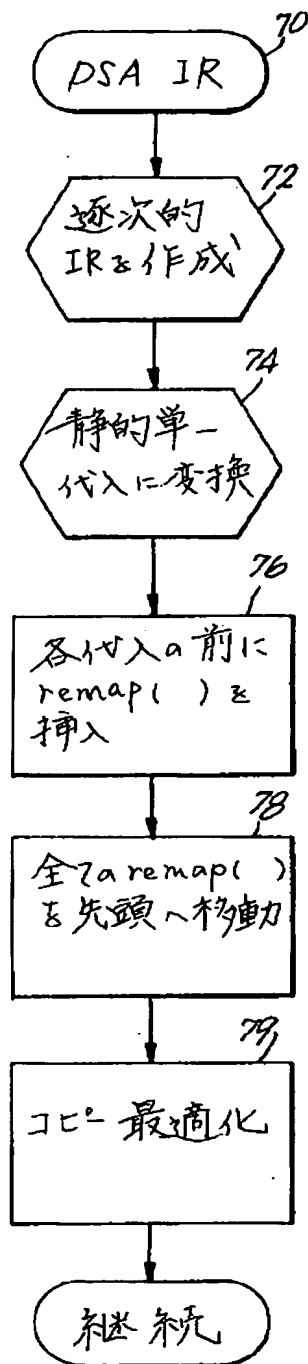
【図2】



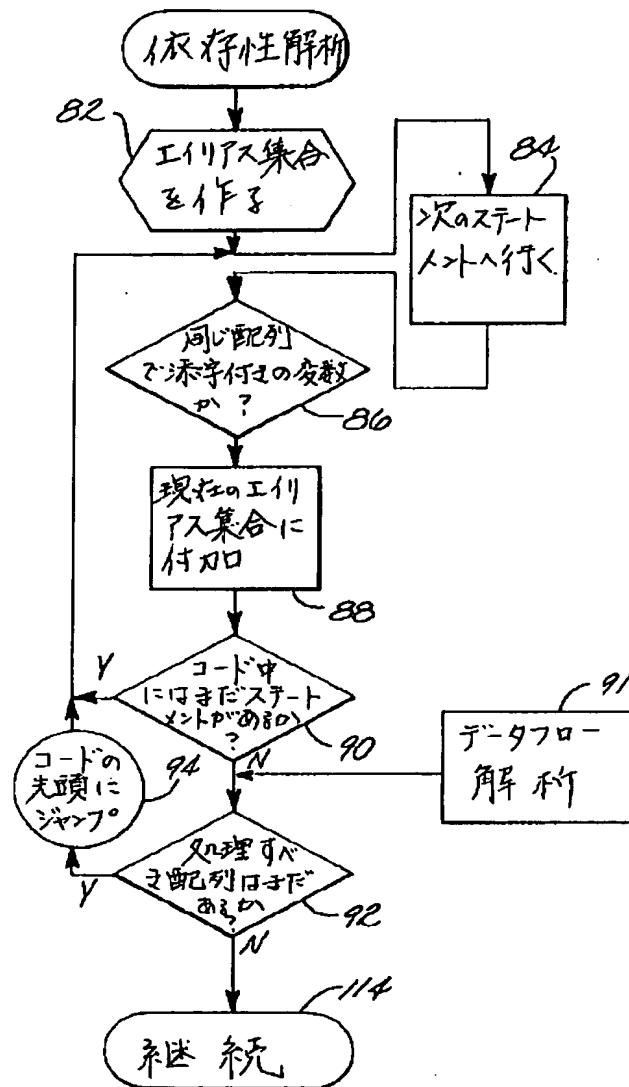
【図3】



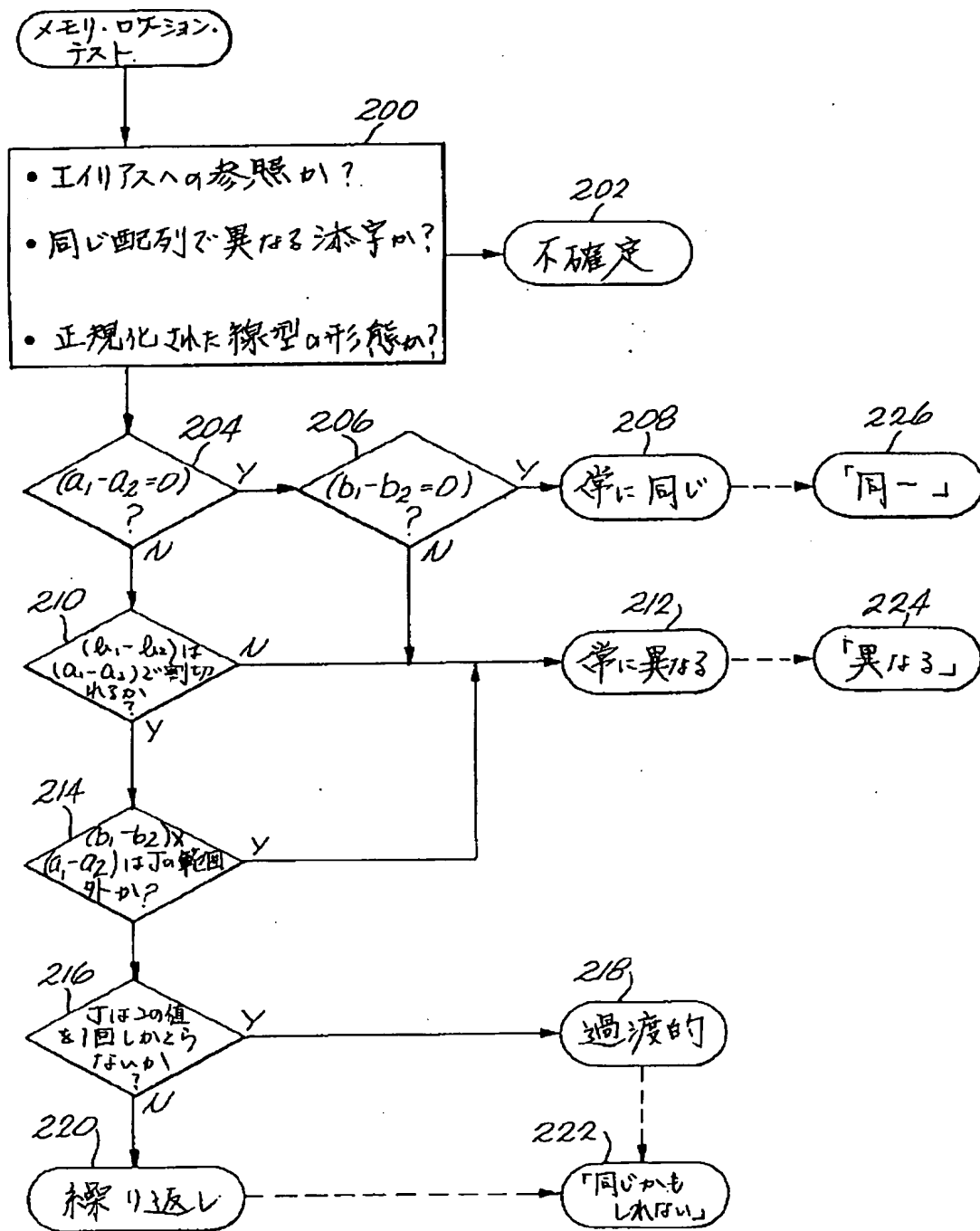
【図4】



【図5】

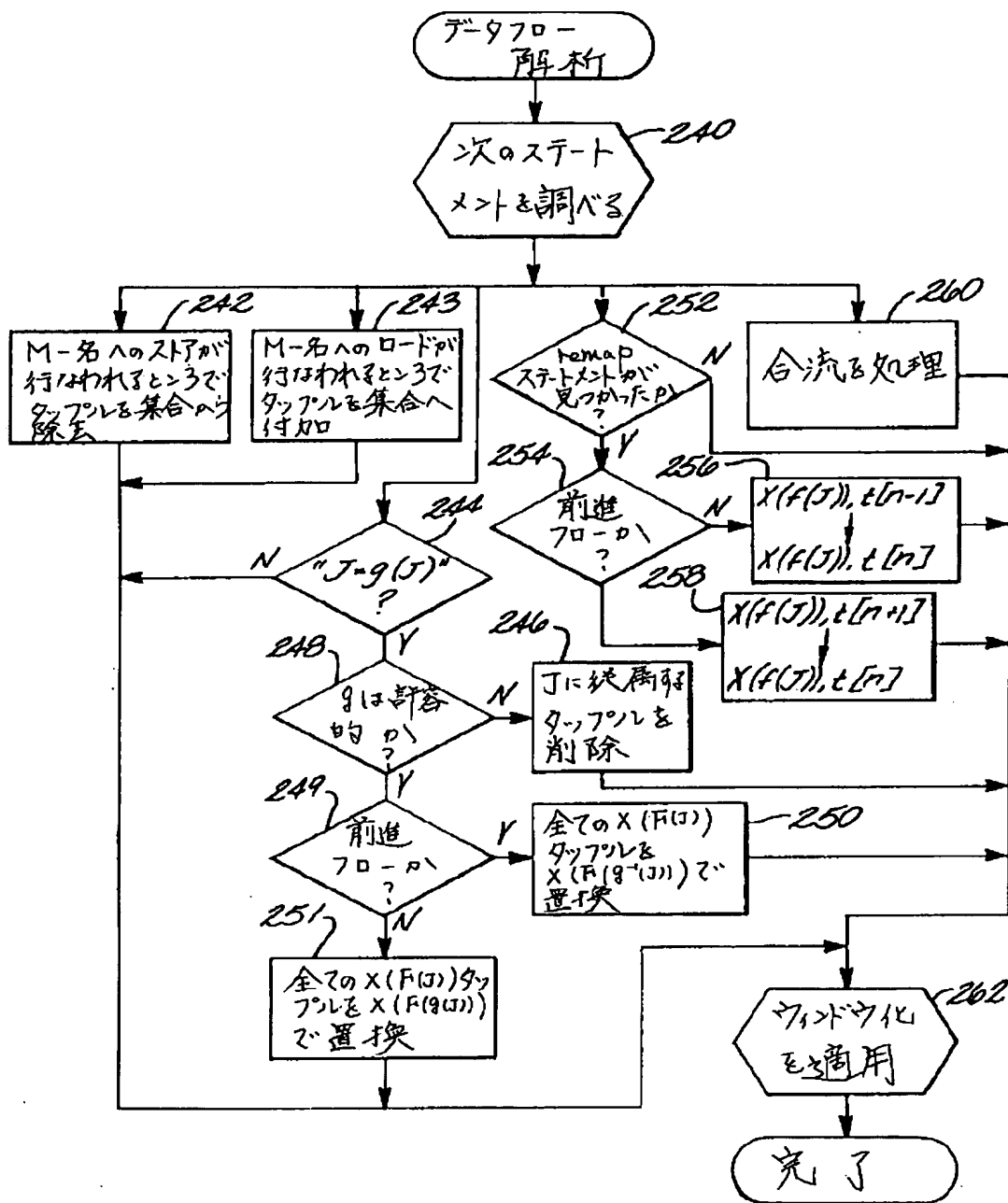


【図6】





【図7】



**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

## **BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☒ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**